Beowulf Case Study

Parallelization of 6S for the Generation of VIIRS Snow/Ice Algorithm Look Up Table on the *Vortex* Linux Cluster

Al Danial
Northrop Grumman Information Technology—IIS
R10/1328, One Space Park, Redondo Beach, CA 90278, USA

March 4, 2004

Abstract

Raytheon/ITSS asked the NPOESS Models and Simulations team to deliver a satellite radiance look-up table (LUT) as part of the VIIRS instrument Snow Cover algorithm work. The computer code that generates the look up table, 6S[1], requires ten hours of run time on a fast Sun workstation [Appendix. A] to produce results for one snow type and VIIRS band combination or "case". Thirty two case runs were required in a time frame of one day. Although look up table generation is amenable to parallel processing, 6S was not written to run on a parallel computer. To relieve the schedule pressure, the Fortran source code to 6S was modified to run on the NPOESS Linux cluster 'vortex'. Run time for a single case was reduced from ten hours on the Sun to eight minutes on the sixty four node Linux cluster, 'vortex'. Run time for all thirty two cases, requiring an estimated 320 hours (thirteen days) on the Sun workstation, has been reduced to less than five hours on the Linux cluster.

Science Background

Measurement of snow cover and snow pack properties is vital to the prediction of water supply and flood potential. Due to the high albedo of snow, measurements of snow cover are also very important for monitoring global climate change. Monitoring of sea ice

[1] 6S is "Second Simulation of the Satellite Signal in the Solar Spectrum" developed by Eric Vermote from the University of Maryland; D. Tanre and J.L. Deuze from Laboratoire d'Optique Atmospherique, URA CNRS 713, Universite' des Sciences et Technologies de Lille; and J.J. Morcrette from the European Centre for Medium Range Weather Forecast, UK.

extent is of vital importance for navigational safety and measurements of sea ice age and extent are valuable for weather forecasting and monitoring of global climate change.

The NPOESS/VIIRS Snow Cover algorithm will result in two products to meet the objectives of providing measurements of snow cover and snow pack properties:
1. Global binary snow/no snow cover mask Environmental Data Record
2. Global snow cover fraction Environmental Data Record

The VIIRS Sea Ice Age algorithms will result in a Sea Ice Age Environmental Data Record.

The VIIRS Snow Cover and Sea Ice Age algorithm is scheduled to be delivered by NGST in December 2004. Both the Snow and Ice algorithms depend on pre-computed LUT's that relate various surface, atmospheric and optical parameters to the observed satellite radiances.

Satellite radiance look up tables and test data sets to evaluate the performance of the algorithms are currently being developed. A provisional LUT is due by the second week of March 2004. An effort has been underway for the past two weeks by Dr. Julienne Stroeve of the University of Colorado, and Dr. Anne Nolin of Oregon State University to generate a provisional LUT table of theoretically computed radiances at satellite level for a given set of surface and atmospheric conditions using the 6S radiative transfer code. For each snow type the 6S code iterates over a range of solar and satellite view angles and atmospheric aerosol loadings.

Parallelization of 6S

Impediments to Parallel Solution

Lihong Wang of the NPOESS Models and Simulations team has run LUT generators in parallel on the NPOESS Linux clusters so it seemed that the 6S-based LUT effort should also be able to take advantage of the clusters. Robert Mahoney explained there are two problems: this version of 6S was not written to run on parallel computers, and the binary input files used by 6S were generated on Sun workstations. Sun and Intel CPU's use different byte orders to store integer and real data, so the binary input files generated by on the Sun workstation cannot be read by the Intel x86-based Linux compute nodes.

Binary I/O

Of the two problems, the binary file incompatibility was first perceived to be the harder to solve. Byte-swapping programs can be written but they take time to develop and debug and are only useful for a given file format. A search on google.com quickly yielded a valuable piece of information: Portland Group compilers—which we use on the NPOESS Linux clusters—have an option, `-byteswapio`, which will generate code for an Intel-based Linux machine that reads and writes binary files intended for "big-endian" workstations.

The 6S Fortran code was compiled on the head node of the vortex cluster using the Portland Group pgf77 compiler and the `-byteswapio` switch. We were pleasantly surprised to see the switch worked as advertised. The Linux executable running on the little-endian Intel P4's read the Sun-generated, big-endian binary files without difficulty.

Parallelizing 6S

Although 6S was developed to run on conventional computers, we knew that by changing the ranges of loops over the independent variables (solar zenith angle, viewer zenith angle, and relative azimuth) we could alter the code to run on a parallel machine by giving multiple processors different combinations of variables to work with. The 6S loop limits are hard-coded in the Fortran source so our options were to either generate a unique executable for each processor, or modify the code to read loop limits from the command line or an input file. Here is actual code from 6S's `main.f`:

```
479 c solar zenith angle loop
480          do 5555 isza=1,40
481           asol=ACOS(cos_sza(isza))*180./pi
482 c viewing zenith angle loop
483           do 6666 ivza=1,40
484            avis=ACOS(cos_sza(ivza))*180./pi
485            phi0=0.
486 c relative azimuth angle loop
487           do 7777 irel=1,37
488            phi0=relaz(irel)
```

The loop ranges of 1 to 40 (`do 5555`), 1 to 40 (`do 6666`), and 1 to 37 (`do 7777`) yield 59,200 combinations of angles. As we have more than 40 compute nodes at our disposal, the outer loops were 'flattened' and their loop variables remapped. Instead of three nested loops over 1 to 40, 1 to 40, and 1 to 37 iterations we had two nested loops: the outer loop going from 1 to 40*40, and the unmodified inner loop on `irel` going from 1 to 37. The loop variables `isza` and `isva` corresponding to the original triple nested loops were remapped as shown on lines 486 and 487:

```
480          nIS = 40
481          nIV = 40
482
483 c solar zenith angle loop + viewing zenith angle loop
484          do 5555 jj=1, nIS*nIV
485
486            isza = int((jj-1) / nIV) + 1
487            ivza = mod((jj-1) , nIV) + 1
488
489            asol=ACOS(cos_sza(isza))*180./pi
490
491            avis=ACOS(cos_sza(ivza))*180./pi
492            phi0=0.
493 c relative azimuth angle loop
494           do 7777 irel=1,37
495            phi0=relaz(irel)
```

Note the absence of the `do 6666` loop.

The outer loop over the new variable `jj` spans 1600 (`nIS*nIV` = 40*40) interations which fits nicely on 64 compute nodes: 64*25 = 1600 so each compute node does 25 passes of the `jj` loop.

At this point we modified the code further by replacing line 484

```
484           do 5555 jj=1, nIS*nIV
```

with

```
484           do 5555 jj=JJ_START, JJ_END
```

where the new variables `JJ_START` and `JJ_END` can either be taken from the command line, read from an input file, or hard-coded for 64 separate instances of the executable. Regardless of the approach the net result would be that compute node 1 would run with

```
do 5555 jj=1, 26
```

compute node 2 would run with

```
do 5555 jj=26,50
```

et cetera, until node 64 which runs with the last 25 cases:

```
do 5555 jj=1576,1600
```

Perl Scripts for Creating Executables, Submitting Jobs, Collecting Results

While it is cleaner to expand the Fortran code to accept command line arguments or file inputs for the `jj` loop start and end values, it was actually much easier to create 64 hard coded executables using Perl commands to edit a file template in-place. The Perl approach meant that no time would be spent writing and testing additional Fortran code needed to handle the command line or input file parsing.

A small Perl script [Appendix B] was written to generate 64 executables where the only difference between executables was the starting and ending values for the `jj` do-loop. Executables were named using the convention "`sixs_`*XX*" where *XX* is a number from 01 to 64. An executable's suffix number matches the compute node to which the executable is submitted. For example `sixs_09` would be submitted to compute node "`msca09`".

A second Perl script [Appendix C] was written to copy to each compute node a compressed tar file containing the 6S binary input files (40 MB when uncompressed) as well as the compute-node-specific executable, then to start the run on that node.

A third Perl script [Appendix D] was written to harvest the output data from each node and reassemble them in correct order into a single file.

Code Modification Effort

It took about three hours discover and test the `-byteswapio` option to the Portland Group compiler, study the relevant portions of the 6S Fortran code, flatten the loops, come up with the index mapping functions, and write and test the three Perl scripts.

Performance

The Linux cluster beats the Sun workstation at two levels: single node performance is better, and more nodes are available to tackle the problem. Note that while both the Sun workstation and the Linux compute nodes have dual CPU's, the 6S code is unable to take advantage of the second processor on either platform.

Single Node Performance

A single node of the Linux cluster runs the 6S code 50% more quickly than the Sun. Timings on unloaded machines to compute eight passes of the inner loop are

Sun	4.54 seconds
Linux	3.06 seconds

The same optimization switch, "-O", was used on both Sun and Portland Group compilers. No additional optimization levels were investigated.

Aggregate Cluster Performance

One compute node on the Linux cluster can find its portion (40*40*37/64 = 925 angle combinations) of the solution for a complete case in 7 minutes 12 seconds. Total solution time for all 64 compute nodes is longer—8 minutes 2 seconds—since it takes 50 seconds to distribute the input data and submit the 64 jobs from the head node, and another 4 seconds to collect and reassemble the results.

Appendix A: Hardware and Compiler Specifications

Sun workstation

Name	lorien
Model	Sun Fire 280R
RAM	8 GB
CPU	dual sparcv9, 1.2 GHz
OS	SunOS v5.8, 64 bit
compiler	Sun WorkShop 6 update 2 FORTRAN 77 5.3 Patch 111691-06 2002/07/23
command	f77 -O -Dsun -c <file.f>

Linux cluster

name	vortex
model	IBM x335
# nodes	64
compiler	Portland Group pgf77 4.0-2
command	pgf77 -O -Dlinux -byteswapio -Wl,-Bstatic -c <file.f>

compute nodes:

RAM	2 GB
CPU	dual Intel P4 Xeon, 2.8 GHz
OS	MSC.Linux (kernel 2.4.18-9)

Appendix B: Perl Script to Create Executables

```
#!/usr/local/bin/perl -w
use strict;
#
# Albert Danial Mar 2, 2004
#
# Create a unique sixs executable for each processor in a cluster.
# An executable has hardcoded loop limits defined in main.f.
# The file main-template.f is used as the template to create the
# individual main.f files.
#

my $nIS      = 40;
my $nIV      = 40;
my $sixs     = "sixs";
my $template = "main-template.f";

die "unable to read $template" unless -r $template;

print "How many processors (nIS=$nIS  nIV=$nIV) ?  ";
chomp(my $nCPU = <STDIN>);
die unless $nCPU =~ /^\d+/ and $nCPU > 0;

my $iter_per_node = int($nIS*$nIV/$nCPU);
my $left_over     = $nIS*$nIV - $nCPU*int($nIS*$nIV/$nCPU);

if ($left_over) {
    printf "%d node(s) get %d iterations; ", $left_over, $iter_per_node + 1;
    printf "%d node(s) get %d iterations;\n", $nCPU-$left_over, $iter_per_node;
} else {
    printf "Even split:  each of %d processors gets %d iterations\n",
           $nCPU, $iter_per_node;
}
sleep 1;

my $n = 0;
foreach my $node (0..($nCPU-1)) {

    unlink $sixs if -f $sixs;

    my $start_jj = $n + 1;
    my $end_jj   = $start_jj + $iter_per_node - 1;
    if ($node < $left_over) {
        ++$end_jj;
    }
    $n = $end_jj;
    printf "node %2d:  %4d -> %4d  (%3d cases)\n",
           $node+1, $start_jj, $end_jj, $end_jj-$start_jj+1;

    system "cp $template main.f";
    system "perl -pi -e 's/JJ_START/$start_jj/' main.f";
    system "perl -pi -e 's/JJ_END/$end_jj/'     main.f";
    system "make";

    die "failed to create $sixs exe" unless -x $sixs;

    my $new_exe = sprintf("sixs_%02d", $node+1);
    rename $sixs, $new_exe;
    printf "Created $new_exe\n";
}
```

Appendix C: Perl Script to Submit Jobs

```
#!/usr/local/bin/perl -w
use strict;
#
# Albert Danial Mar 2, 2004
#
# Submit jobs to each node for which there is a sixs executable.
# The executables must have been created earlier with the script
# Generate_sixs_exes
#

opendir(DIR, ".");
my @exes = grep(/^sixs_\d\d$/, readdir(DIR));
closedir(DIR);
die "Could not find any sixs_xx executables in this directory\n" unless @exes;

print "\n", join(" ", @exes), "\n";
print "\nUse these executables?  y/n\n";
chomp(my $ans = <STDIN>);
die unless $ans eq "y";

foreach my $exe (@exes) {

    my $node_number = $exe;
    $node_number    =~ s/sixs_//;
    my $node        = "msca" . $node_number;

    print "submitting $exe to $node\n";

    system "rcp sixs_data.tar.bz2 $node:/scratch/s116493";
    system "rcp $exe              $node:/scratch/s116493";
    system "rsh $node \"cd /scratch/s116493; tar jxf sixs_data.tar.bz2\"";
    system "rsh -n $node \"cd /scratch/s116493; nohup ./$exe \" \& ";

}
```

Appendix D: Perl Script to Collect Results

```
#!/usr/local/bin/perl -w
use strict;
#
# Albert Danial Mar 2, 2004
#

my $solution_file = "/home/s116493/mahoney/reflect.dat_M1_1000_od50";

opendir(DIR, ".");
my @exes = grep(/^sixs_\d\d$/, readdir(DIR));
closedir(DIR);
die "Could not find any sixs_xx executables in this directory\n" unless @exes;

print "\n", join(" ", sort @exes), "\n";
print "\nGet data generated by these executables?  y/n\n";
chomp(my $ans = <STDIN>);
die unless $ans eq "y";

unlink $solution_file if -f $solution_file;

foreach my $exe (sort @exes) {

    my $node_number = $exe;
    $node_number    =~ s/sixs_//;
    my $node        = "msca" . $node_number;

    print "Getting solution from $node\n";

    system "rcp $node:/scratch/s116493/reflect.dat_M1_1000_od50 refl.$node";
    system "cat refl.$node >> $solution_file";
    unlink "refl.$node";

}
```